PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/40 — A1

(11) International Publication Number: WO 99/39261

(43) International Publication Date: 5 August 1999 (05.08.99)

(21) International Application Number: PCT/US98/21406

(22) International Filing Date: 9 October 1998 (09.10.98)

(30) Priority Data:
08/948,109 9 October 1997 (09.10.97) US

(71) Applicant: THE LEARNING COMPANY [US/US]; 1 Athenaeum Street, Cambridge, MA 02142 (US).

(72) Inventor: TOMIC, Ratko, V.; 7 Tufts Road, Lexington, MA 02173 (US).

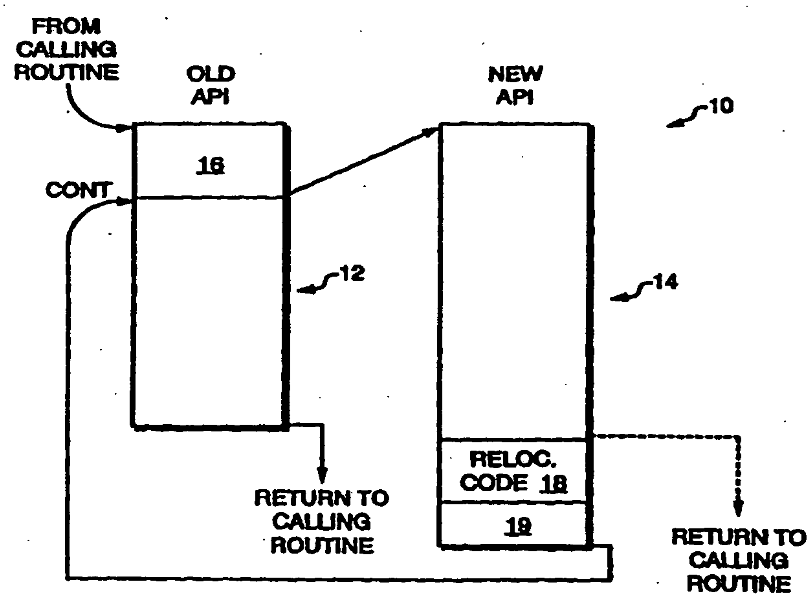
(74) Agents: MUIRHEAD, Donald, W. et al.; Foley, Hoag & Eliot LLP, One Post Office Square, Boston, MA 02109 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, HR, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: WINDOWS API TRAPPING SYSTEM

(57) Abstract



Supplementing a software routine loaded in computer memory includes loading an additional routine into the computer memory, providing relocated opcodes by relocating a number of bytes from a relocatable portion of the software routine to another memory location where the number of bytes corresponds to an integral number of instructions of the relocatable portion, causing program control to flow from the additional routine to the relocated opcodes, causing program control to flow from the relocated opcodes to a memory address immediately following the relocatable portion, and causing program control to flow from the relocatable portion to the additional routine. Causing program control to flow from the additional routine to the relocated opcodes may include placing the relocated opcodes at a location in the computer memory that immediately follows the additional routine. Causing program control to flow from the relocated opcodes to the memory address immediately following the relocatable portion may include placing a program control instruction at a location in the computer memory immediately following the relocated opcodes. Causing program control to flow from the relocatable portion to the additional routine may include placing a program control instruction at a memory location corresponding to a source address of the relocatable portion of the software routine.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece	ML	Mali	TR	Turkey
BG	Bulgaria	HU	Hungary	MN	Mongolia	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MR	Mauritania	UA	Ukraine
BR	Brazil	IL	Israel	MW	Malawi	UG	Uganda
BY	Belarus	IS	Iceland	MX	Mexico	US	United States of America
CA	Canada	IT	Italy	NE	Niger	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NL	Netherlands	VN	Viet Nam
CG	Congo	KE	Kenya	NO	Norway	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NZ	New Zealand	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	PL	Poland		
CM	Cameroon	KR	Republic of Korea	PT	Portugal		
CN	China	KZ	Kazakstan	RO	Romania		
CU	Cuba	LC	Saint Lucia	RU	Russian Federation		
CZ	Czech Republic	LI	Liechtenstein	SD	Sudan		
DE	Germany	LK	Sri Lanka	SE	Sweden		
DK	Denmark	LR	Liberia	SG	Singapore		
EE	Estonia						

WINDOWS API TRAPPING SYSTEM

Cross-Reference to Related Applications

This application is based on U.S. provisional patent application No. 60/028,339, filed on October 11, 1996.

Background of The Invention

1. Field of the Invention

This application relates to the field of software and more particularly to the field of managing aspects of an interface between software and the underlying operating system.

2. Description of Related Art

Many conventional operating systems provide a formalized Application Programming Interface (API) that allows application programmers to make calls to software routines that perform a variety of system-wide functions. Using the API facilitates writing application programs by decreasing the amount of code that application programmers need to provide and, at the same time, providing standardization of routines that are used by many of the applications.

However, in some instances, it is necessary to modify an API call in order to, for example, perform specialized functions that are not provided by the operating system or to keep track of certain types of API calls. For some operating systems, it is not difficult to intercept and monitor API calls. For example, under MS-DOS, most APIs use interrupts and are thus easy to intercept. In other instances, intercepting and trapping API calls can be challenging for an application programmer. For example, in some Microsoft Windows Environments (Win16 and Win32), the APIs use exported functions which are connected to the application at application load time via static or dynamic linking. This linkage process is done by the internal OS routines and undocumented data structures which are usually inaccessible to outside (non-Microsoft) software developers. In addition, newer

versions of the Windows OS actively thwart API intercepting methods described in the programming literature for the earlier versions of the OS. In some instances, a technological race is unfolding between the operating system developers and the third party (application) developers who need to provide OS/applications enhancements unforseen (or undesired) by the OS developers.

A method for intercepting Windows 3.x APIs based on patching the entry point of the API function with a JMP instruction is known. In this system, the interceptor first obtains the address of the target API function via a call to GetProcAddress(). Note that this step has been actively thwarted by the OS in Win95 for many key API functions, although within months the workarounds for much of the thwarting attempts have been published. Following getting the procedure address, the interceptor removes the write-protection of the obtained memory address (which is a code address, thus it is set by the OS as read/execute-only). Following this, the interceptor patches the API entry point with JMP InterceptSrv instruction, where InterceptSrv is a service function in the interceptor's code that monitors and/or processes the API calls.

After the interceptor has set up the API call in the manner described above, an application or OS calls the API entry point. The JMP InterceptSrv instruction at the entry point address transfers control to the InterceptSrv() function and the InterceptSrv() processes the call (accessing as necessary the function arguments on the stack) and, upon completion, either returns control to the caller or passes control to the original API function. Returning control to the caller is straightforward. Passing control to the API, on the other hand, requires that the InterceptSrv() remove the patched in JMP InterceptSrv from the API entry point and restore the original opcode bytes (the original five bytes that were present in the API before the patch was inserted). Then, the InterceptSrv() pushes all original arguments from the caller's stack on to the current stack (interceptor's stack). Finally, the InterceptSrv() calls the original API function. When the original API returns, the InterceptSrv() saves the return value into a local variable, the

InterceptSrv() reinserts the JMP InterceptSrv patch at the API entry point, and the InterceptSrv() returns the saved return value to the original caller.

The technique described above is used in many commercial application enhancers. However, it has many drawbacks. For one thing, the division of labor between the set up portion and the run time portion is highly inefficient since the set up portion is executed only once (at the interceptor's load time) while the run time portion is executed many times (from the load time onward). In a more efficient system, as much of the work as possible would be shifted from the run time portion to the set up portion. In addition, for a processor running MS Windows on an Intel processor, the necessary replacement of the JMP Intercept opcodes requires a write operation of five bytes, which can not be done in a single processor instruction. Hence, the replacement leaves a short interval of instability between the two write instructions, during which the API entry point has invalid instructions and a hardware interrupt at that moment could initiate reentry into the API, which will probably crash the system.

On the other hand, disabling the interrupts in Windows application mode (CPU ring 3 code where these actions are occurring) is a highly expensive operation due to system control over the CPU interrupt flag, which triggers an elaborate exception process (in CPU ring 0). The complex ring transition and the exception process, which may take as much or more time than all the rest of processing above, would occur twice. Therefore most commercial interceptors (as well as the published code) choose the tradeoff with the instability allowed, in order not to pay the disproportionate performance cost associated with disabling interrupts.

Another disadvantage of the technique described above is that, during certain time intervals, another call to the monitored API will be missed by the interceptor. Since many Windows APIs perform checks on the task queue within and can (and often do) switch to another thread/task, the missing of intercepts is a real problem for interceptors which

require processing on every call to the API (especially those implementing security features). Also, the overwriting of the API entry point on every call is unsafe when multiple interceptors exist on the same system. For example, while the system is processing the original API call, a switch to another task or thread can (and often will) occur. If the second task inserts its own intercept for the same API (or spawns a program which does that), then return to the first interceptor will destroy the new intercept, thus permanently disabling operation of the new intercept. Additionally, if the first intercept unloads, followed by an unload of second intercept, then the API entry will be left pointing to the non-existent first intercept and the system will crash when the API is invoked.

Another disadvantage of the technique described above is that some Windows API functions (e.g. memory allocation and protection) are sensitive to the source of the call, i.e. the Windows API code will check where the call is made from and, based on knowledge of Microsoft's sources of calls, will work differently if called from third party applications as opposed to particular Microsoft sources. This behavior is, among other reasons, related to the active thwarting of the third party interceptors mentioned above. Since the intercept technique described above changes the original source of the API call and makes the interceptor appear to Windows as the source of the API call, the Windows API processing may operate differently, often malfunctioning in a way that leads to instabilities and system crashes. This makes the technique described above unsuitable for intercepting some of the Windows APIs.

Summary Of The Invention

According to the present invention, supplementing a software routine loaded in computer memory includes loading an additional routine into the computer memory, providing relocated opcodes by relocating a number of bytes from a relocatable portion of the software routine to an other memory location where the number of bytes corresponds to an integral number of instructions of the relocatable portion, causing program control

to flow from the additional routine to the relocated opcodes, causing program control to flow from the relocated opcodes to a memory address immediately following the relocatable portion, and causing program control to flow from the relocatable portion to the additional routine.

Causing program control to flow from the additional routine to the relocated opcodes may include placing the relocated opcodes at a location in the computer memory that immediately follows the additional routine. Causing program control to flow from the relocated opcodes to the memory address immediately following the relocatable portion may include placing a program control instruction at a location in the computer memory immediately following the relocated opcodes. The program control instruction may be an unconditional jump instruction. Causing program control to flow from the relocatable portion to the additional routine may include placing a program control instruction at a memory location corresponding to a source address of the relocatable portion of the software routine. The program control instruction may be an unconditional jump instruction. Providing the relocated opcodes may include relocating a number of bytes that is at least equal to an amount of bytes required for the unconditional jump instruction.

Following providing the relocated opcodes, it is possible to resolve any opcodes contained therein that reference relative displacements between the relocated opcodes and opcodes contained in the software routine. The software routine and additional routine may be API's that run under the Microsoft Windows operating system. The additional routine may be configured to load at a predetermined address in the computer memory.

According further to the present invention, supplementing a Windows API loaded in computer memory includes loading an additional routine into the computer memory, providing relocated opcodes by relocating a number of bytes from a relocatable portion of the API to an other memory location where the number of bytes corresponds to an integral number of opcodes of the relocatable portion, causing program control to flow

from the additional routine to the relocated opcodes, causing program control to flow from the relocated opcodes to a memory address immediately following the relocatable portion, and causing program control to flow from the relocatable portion to the additional routine. According further to the present invention, a software program that supplements a Windows API loaded in computer memory includes an additional routine that is loaded into the computer memory, first means for providing relocated opcodes by relocating a number of bytes from a relocatable portion of the API to an other memory location where the number of bytes corresponds to an integral number of instructions of the relocatable portion, second means, coupled to the first means and to the additional routine, for causing program control to flow from the additional routine to the relocated opcodes, third means, coupled to first means and to the API, for causing program control to flow from the relocated opcodes to a memory address immediately following the relocatable portion, and fifth means, coupled to the API and to the additional routine, for causing program control to flow from the relocatable portion to the additional routine.

In the intercept install phase, the technique described herein disassembles the target API entry code and relocates (based on the semantics of the instructions found there) the whole instructions from the API entry into the interceptor's memory. Then, in the intercept operation phase, when control needs to be passed to the original API function, instead of having to swap back and forth the overlayed API entry opcodes, the intercept simply passes control to the second (relocated) copy of the API entry code, which, upon completion, passes control to the next section of the original API code (the section which follows the relocated section).

This method thus shifts the division of labor heavily toward the install phase of the intercept, relieving the active phase of the intercept by eliminating many of the steps described in connection with the prior art system and the performance, safety and system stability drawbacks associated with them.

The technique described herein has many advantages over conventional trapping systems. One advantage is that all trapping work, except for a minimum amount of work necessary to transfer control to the interceptor and back, is done only once at install time, relieving the performance burden from the run time activity of the interceptor. In addition, no opcode swapping is done during the existence and activation of the traps. This eliminates performance, stability and safety drawbacks resulting from the activity found in some conventional systems. Furthermore, since the execution of the some of the trapping code occurs on the stack of the original caller, there is no need to copy API function arguments to the interceptor's stack. By executing much of the code on the original caller's stack, the source of the call will appear to the Windows API as if it came from the original caller, therefore resolving problems associated with Windows behaving differently depending on the identity of the caller. By not copying API entry opcodes back and forth at each API call, no window of system instability is created. Instead, time-consuming precautions (e.g., disabling interrupts) occurs once at intercept install time.

There are additional advantages. Since the trap opcodes are never removed during processing of the original API, the possibility of missing API calls is significantly decreased, as described above. Therefore, the system described herein is an excellent choice for situations where the interceptor must see all of the API calls to the target API function to operate properly or reliably. The problem of multiple interceptors setting traps while processing of the trap is going on is resolved, since the new interceptor will always see fixed opcodes at the API entry point, thus the second interceptor will not be forcibly disabled, as in conventional techniques. Also, since the setting and removal of the traps occur only once at load/unload time of interceptor, the problem of dangling intercept (with target of JMP InterceptSrv already unloaded) can be avoided since the interceptor can afford more detailed, time consuming, checks for safe removal of the intercepts (e.g. by refusing to unload itself if it is not the last interceptor). Doing these checks in the old trapping system is not only time consuming on every API call, but it is in most cases extremely difficult, if not inherently impossible, since it would require that the interceptor

refuse execution of the original API call, which will cause malfunction in the calling application. In addition, the system described herein protects against unauthorized canceling of installed API security functions since canceling a new API task installed using the technique described herein, without restoring the original opcodes of the original API, will likely cause the system to crash.

Brief Description Of Drawings

FIG. 1 is a diagram illustrating a relationship between an old API and a new API according to the present invention.

FIG. 2 is a flow chart showing steps that are performed to install the new API according to the present invention.

FIG. 3 is a flow chart illustrating steps that are performed to remove the new API that is installed using the steps of FIG. 2.

Detailed Description of the Preferred Embodiment(s)

Refer to FIG. 1, a diagram 10 illustrates a relationship between an old API 12 and a new API 14. The old API 12 represents an existing API provided with an operating system, such as MS Windows 95. The new API 14 represents an API that is provided for use in connection with, for example, an applications program. As discussed in more detail below, the new API 14 can be executed instead of the old API 12 or can be executed in addition to the old API 12. Note that, in some instances, the new API 14 and/or the old API 12 may be referred to herein as a "routine". However, the term "routine" should not be understood as referring to a single, unitary, block of code but, instead, should be understood to refer to a collection of code that may be provided in a plurality of blocks that may make calls or jumps therebetween. Note also that the specific functionality provided by the new API 14 is a design choice but may, in some instances, including saving and restoring registers used by the caller and/or the old API 12.

In order to execute the new API 14, the old API 12 is patched with an unconditional jump instruction 16 that transfers control from the old API 12 to the beginning of the new API 14. If the new API 14 is executed instead of the old API 12 (i.e., the old API 12 is not to be executed), then a return to the calling routine occurs at the end of the new API 14, as indicated by the dotted line shown at the end of the new API 14. Note that, as will be apparent to one of ordinary skill in the art, other suitably equivalent control flow instructions may be used in place of the unconditional jump instruction 16.

If the old API 12 will be executed in addition to the new API 14, then opcodes that were located at a relocatable portion of the old API 12 (in this case the beginning of the old API 12) become relocated code 18 that is placed at the end of the new API 14. As discussed in more detail hereinafter, any relative offsets between opcodes within the relocated code 18 and opcodes in the remainder of the old API 12 are adjusted, as appropriate. Note that, as will be apparent to one of ordinary skill in the art, it is possible to place the relocated code 18 at an other portion of memory and then use an appropriate control flow instruction at the end of the new API 14 to transfer program control from the new API 14 to the relocated code 18.

Immediately following the end of the relocated code 18 is an unconditional jump instruction 19 that transfers program control to the portion of the old API 12 immediately following the relocatable portion of the old API 12, marked on the diagram 10 with the address "CONT". Thus, if both the old API 12 and the new API 14 are to be executed, then the calling routine calls the old API 12 which jumps, via the jump instruction 16, to the beginning of the new API 14 which then executes and, at the end thereof, executes the opcodes of the relocated code 18 followed by the jump instruction 19 that jumps back to the remainder of the old API 12. Note that the relocated code 18 and the portion of the old API 12 beginning at the CONT address constitute the entirety of the old API 12. Also note that, as will be apparent to one of ordinary skill in the art, other suitably

equivalent control flow instructions may be used in place of the unconditional jump instruction 19.

Referring to Fig. 2, a flow chart 20 illustrates steps for making patches that cause execution of the new API 14 when an application program or the operating system calls the old API 12. Processing begins at a first step 22, where a byte from the beginning of the old API 12 is fetched. Following the step 22 is a test step 24 which determines if a whole instruction (as opposed to a partial instruction) has been fetched. This determination is made in a conventional fashion by, for example, disassembling the fetched bytes. Note that it is necessary to fetch an integral number of instructions from the relocatable portion of the old API 12 since it is not possible to execute a partial instruction.

If it is determined at the test step 24 that one or more whole instructions have not been fetched, then control passes from the test step 24 back to the step 22 to fetch another byte. Otherwise, if an integral number of instructions have been fetched, then control passes from the test step 24 to a test step 26 which determines if enough bytes have been fetched from the relocatable portion of the old API 12 to accommodate the unconditional jump instruction 16 that transfers control from the old API 12 to the new API 14. In some embodiments, the required number of bytes is five. However, the test step 24 preceding the test step 26 makes it possible that a number of bytes greater than five will have been fetched since it is necessary that a number of bytes corresponding to an integral number of instructions be fetched from the relocatable portion of the old API 12.

If it is determined at the test step 26 that enough bytes have not been fetched, then control passes from the test step 26 back to the step 22 where another byte is fetched. Otherwise, if enough bytes have been fetched, then control passes from the test step 26 to a step 28. Note that it is not possible to reach the test step 26 without having fetched a number of bytes corresponding to an integral number of instructions. This is because it is

not possible to execute the test step 26 without having passed the test at the step 24, which determines that the number of fetched bytes corresponds to a whole number of instructions.

At the step 28, the bytes that have been fetched are moved from the relocatable portion of the old API 12 to the end of the new API 14 and any opcodes in the relocated code 18 that refer to relative offsets are resolved. Note that opcodes in the relocated code 18 that contain a relative offset, such as a jump relative or a call relative, may need to be modified when the opcodes are relocated. Also note that, the relative positions within memory of the old API 12 and the new API 14 should not change after the API's 12, 14 are loaded in memory.

Following the step 28 is a step 30 where the unconditional jump instruction 19 is added to the end of the new API 14. As discussed above, the unconditional jump instruction 19 causes control to return back to the portion of the old API 12 that follows the relocatable portion of the old API 12. Following the step 30 is a step 32 where the unconditional jump instruction 16 is added to the beginning of the old API 12 so that when an application program or the operating system calls the old API 12, the unconditional jump instruction 16 from the old API 12 to the new API 14 will be executed.

As discussed above, it is possible that the new API 14 entirely replaces the old API 12 so that no part of the old API 12 needs to be executed once the new API 14 has been provided. In that case, an alternative patch is provided. As shown in FIG. 2, control passes from the test step 26 to a step 34 where a return instruction is added to the end of the new API 14 (if a return instruction is not already found at the end thereof). Following the step 34, control passes to the step 32 where the unconditional jump instruction 16 is added to the relocatable portion of the old API so that a call to the old API 12 will cause program control to flow from the old API 12 to the new API 14.

Note that, in some instances, it may be unadvisable to relocate the code that is at the beginning of the old API 12. For example, there may be other instructions within the old API 12 that reference the code located at the beginning thereof. In those cases, it is possible to use other portions of the old API 12 as the relocatable portion. For example, it would be possible to relocate bytes following the first N microprocessor instructions, in the manner described above, and replace the relocated bytes with the unconditional jump instruction 16.

The code that executes the patching step illustrated by the flow chart 20 may be written in a conventional computer source language, such as C++, and compiled in a conventional manner similar to compilation of other Microsoft Windows DLL's. In some instances, the preferred base address of the new API 14 may be set to a value that will cause the new API 14 to always load at the same address for all the processes which use the new API 14. Thus, the new API 14 may be shared so that the new API 14 is loaded in memory only once, even when used by multiple processes.

Refer to FIG. 3, a flow chart 40 illustrates steps that are performed when a process that uses the new API 14 is removed from memory. Note that, under the Windows environment, a special routine (MS Main) is called when a process is removed from memory. The MS Main routine provides the application with an opportunity to do cleanup including, in this instance, restoring the old API 12.

Processing begins at a first test step 42 where it is determined if the process being removed is the last process that uses the new API 14. If not, then the old API 12 and the new API 14 are not modified and no cleanup is done, since the new API 14 must remain to be used by the other processes. Otherwise, control passes from the test step 42 to a step 44 where the relative instructions of the relocated code 18 are modified back to the original state prior to restoring the relocated code into the old API 12. Following the step 44 is a step 46 where the relocated code 18 is restored into the old API 12, thus

overriding the unconditional jump instruction 16 that was provided to the old API 12 when the old API 12 was patched. Once the relocated code 18 is restored to the old API 12, then a call to the old API 12 will not result in execution of the new API 14.

Note that, although the invention has been illustrated herein using APIs with the Windows operating system, it would be straight-forward for one of ordinary skill in the art to adapt the system described herein to other operating systems and other types of routines.

While the invention has been disclosed in connection with the preferred embodiments shown and described in detail, various modifications and improvements thereon will become readily apparent to those skilled in the art. Accordingly, the spirit and scope of the present invention is to be limited only by the following claims.

Claim(s)

1. A method of supplementing a software routine loaded in computer memory, comprising:

(a) loading an additional routine into the computer memory;

(b) providing relocated opcodes by relocating a number of bytes from a relocatable portion of the software routine to an other memory location, the number of bytes corresponding to an integral number of instructions of the relocatable portion;

(c) causing program control to flow from the additional routine to the relocated opcodes;

(d) causing program control to flow from the relocated opcodes to a memory address immediately following the relocatable portion; and

(e) causing program control to flow from the relocatable portion to the additional routine.

2. A method, according to claim 1, wherein causing program control to flow from the additional routine to the relocated opcodes includes placing the relocated opcodes at a location in the computer memory that immediately follows the additional routine.

3. A method, according to claim 1, wherein causing program control to flow from the relocated opcodes to the memory address immediately following the relocatable portion includes placing a program control instruction at a location in the computer memory immediately following the relocated opcodes.

4. A method, according to claim 3, wherein the program control instruction is an unconditional jump instruction.

5. A method, according to claim 1, wherein causing program control to flow from the

relocatable portion to the additional routine includes placing a program control instruction at a memory location corresponding to a source address of the relocatable portion of the software routine.

6. A method, according to claim 5, wherein the program control instruction is an unconditional jump instruction.

7. A method, according to claim 6, wherein providing the relocated opcodes includes relocating a number of bytes that is at least equal to an amount of bytes required for the unconditional jump instruction.

8. A method, according to claim 1, further comprising:

(f) following providing the relocated opcodes, resolving any opcodes contained therein that reference relative displacements between the relocated opcodes and opcodes contained in the software routine.

9. A method, according to claim 1, wherein the software routine and additional routine are API's that run under the Microsoft Windows operating system.

10. A method, according to claim 9, wherein the additional routine is configured to load at a predetermined address in the computer memory.

11. A method of supplementing a Windows API loaded in computer memory, comprising:

(a) loading an additional routine into the computer memory;

(b) providing relocated opcodes by relocating a number of bytes from a relocatable portion of the API to an other memory location, the number of bytes corresponding to an integral number of instructions of the relocatable portion;

(c) causing program control to flow from the additional routine to the

relocated opcodes;

(d) causing program control to flow from the relocated opcodes to a memory address immediately following the relocatable portion; and

(e) causing program control to flow from the relocatable portion to the additional routine.

12. A software program that supplements a Windows API loaded in computer memory, comprising:

an additional routine that is loaded into the computer memory;

first means for providing relocated opcodes by relocating a number of bytes from a relocatable portion of the API to an other memory location, the number of bytes corresponding to an integral number of instructions of the relocatable portion;

second means, coupled to the first means and to the additional routine, for causing program control to flow from the additional routine to the relocated opcodes;

third means, coupled to first means and to the API, for causing program control to flow from the relocated opcodes to a memory address immediately following the relocatable portion; and

fifth means, coupled to the API and to the additional routine, for causing program control to flow from the relocatable portion to the additional routine.

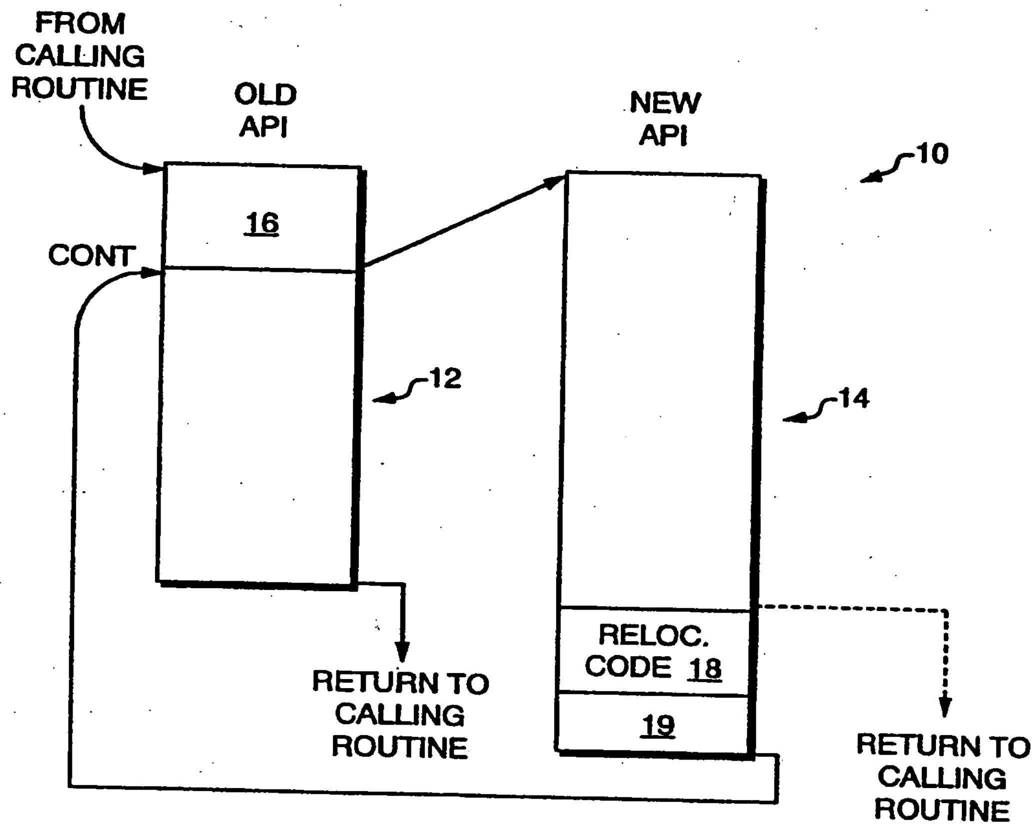

FROM
CALLING
ROUTINE
OLD
API
NEW
API
10
16
CONT
12
14
RETURN TO
CALLING
ROUTINE
RELOC.
CODE 18
19
RETURN TO
CALLING
ROUTINE

FIG. 1

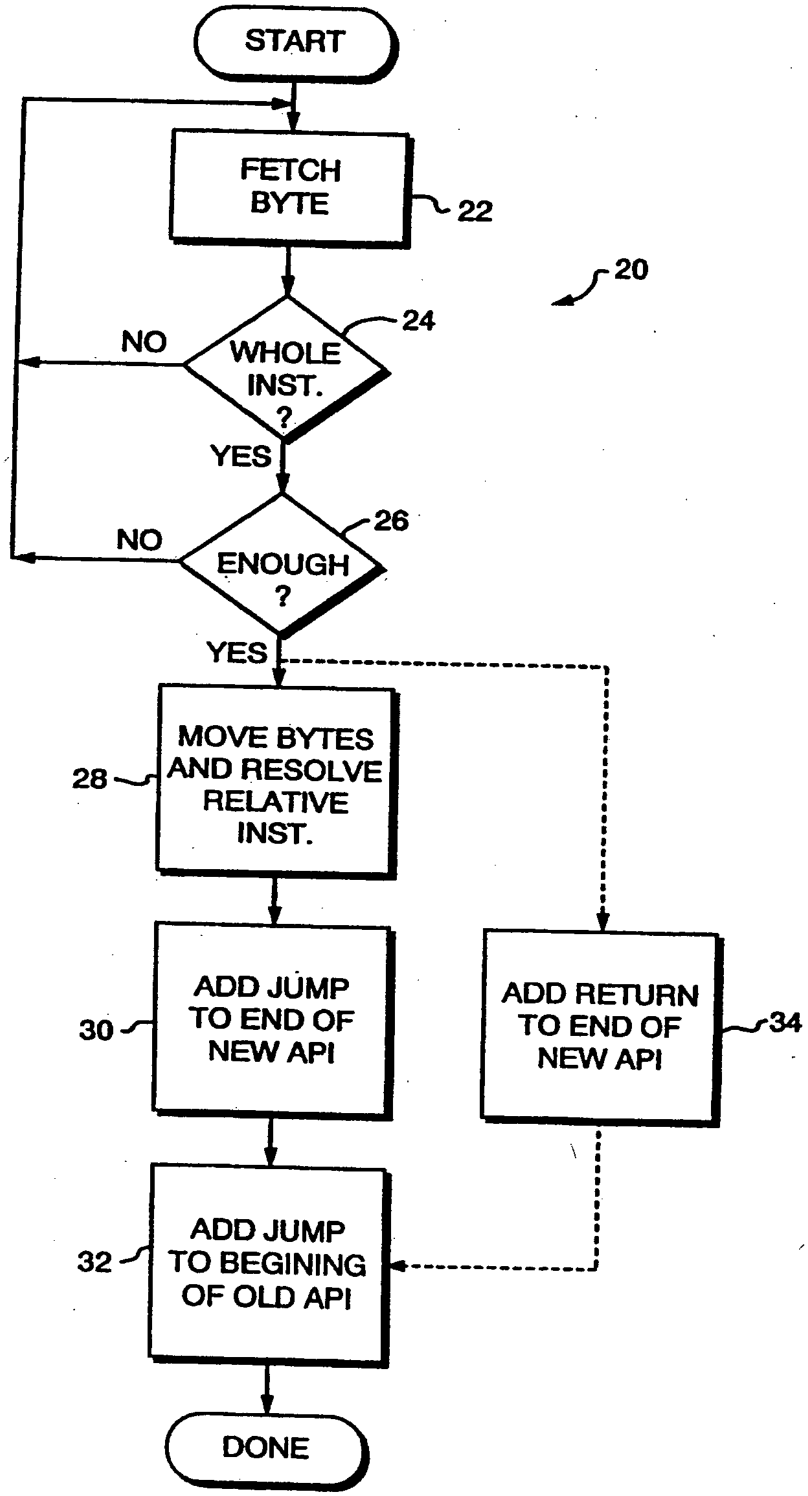

START
FETCH BYTE
22
20
24
NO
WHOLE INST. ?
YES
26
NO
ENOUGH ?
YES
MOVE BYTES AND RESOLVE RELATIVE INST.
28
ADD JUMP TO END OF NEW API
30
ADD RETURN TO END OF NEW API
34
ADD JUMP TO BEGINING OF OLD API
32
DONE

FIG. 2

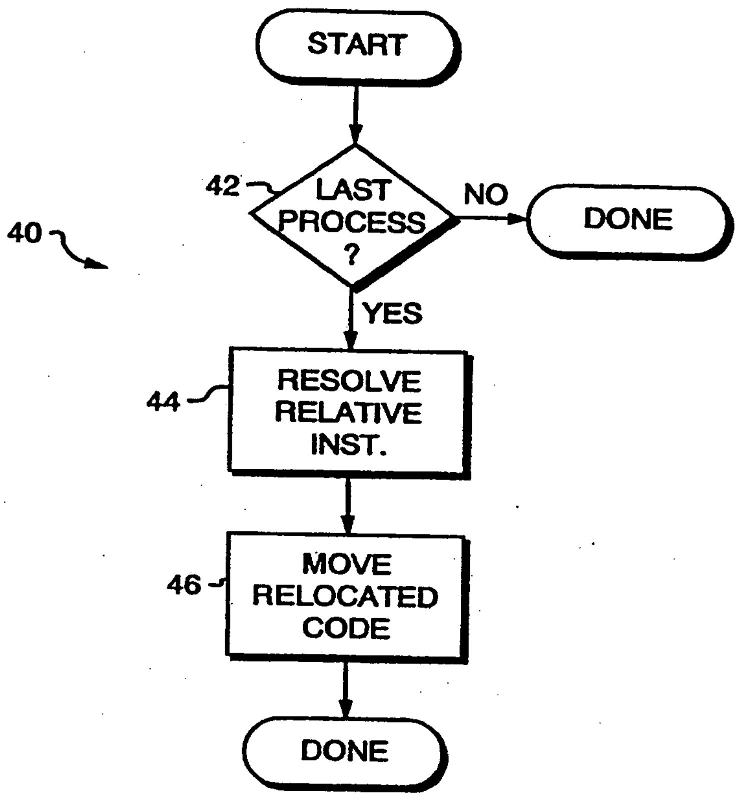

START
42
LAST PROCESS ?
NO
DONE
40
YES
44
RESOLVE RELATIVE INST.
46
MOVE RELOCATED CODE
DONE

FIG. 3

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 98/21406

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/40

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	WO 94 27221 A (SIEMENS AG) 24 November 1994 see the whole document	1-12
Y	EP 0 665 496 A (SUN MICROSYSTEMS INC) 2 August 1995 see the whole document	1-12
A	WO 93 00633 A (PURE SOFTWARE INC) 7 January 1993 see page 4, line 3 - line 25 see page 8, line 35 - page 13, line 17 see figures 2-4	1-12

-/--

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search

14 May 1999

Date of mailing of the international search report

28/05/1999

Name and mailing address of the ISA
European Patent Office, P.B. 5818 Patentlaan 2
NL - 2280 HV Rijswijk
Tel. (+31-70) 340-2040, Tx. 31 651 epo nl,
Fax: (+31-70) 340-3016

Authorized officer

Fonderson, A

1

INTERNATIONAL SEARCH REPORT

In. ...tional Application No
PCT/US 98/21406

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
A	MATT PIETREK: "Intercepting API Functions in Win32" PC MAGAZINE, vol. 13, no. 19, 11 August 1994, pages 307-312, XP002102766 NEW YORK US see the whole document	1-12

1

INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No
PCT/US 98/21406

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 9427221 A	24-11-1994	DE 4315944 A	17-11-1994
		EP 0698239 A	28-02-1996
		JP 8510342 T	29-10-1996
EP 0665496 A	02-08-1995	US 5581697 A	03-12-1996
		JP 8036488 A	06-02-1996
		US 5675803 A	07-10-1997
WO 9300633 A	07-01-1993	US 5193180 A	09-03-1993
		AU 2188792 A	25-01-1993
		CA 2111958 A	07-01-1993
		EP 0591360 A	13-04-1994
		US 5535329 A	09-07-1996
		US 5835701 A	10-11-1998
		US 5335344 A	02-08-1994